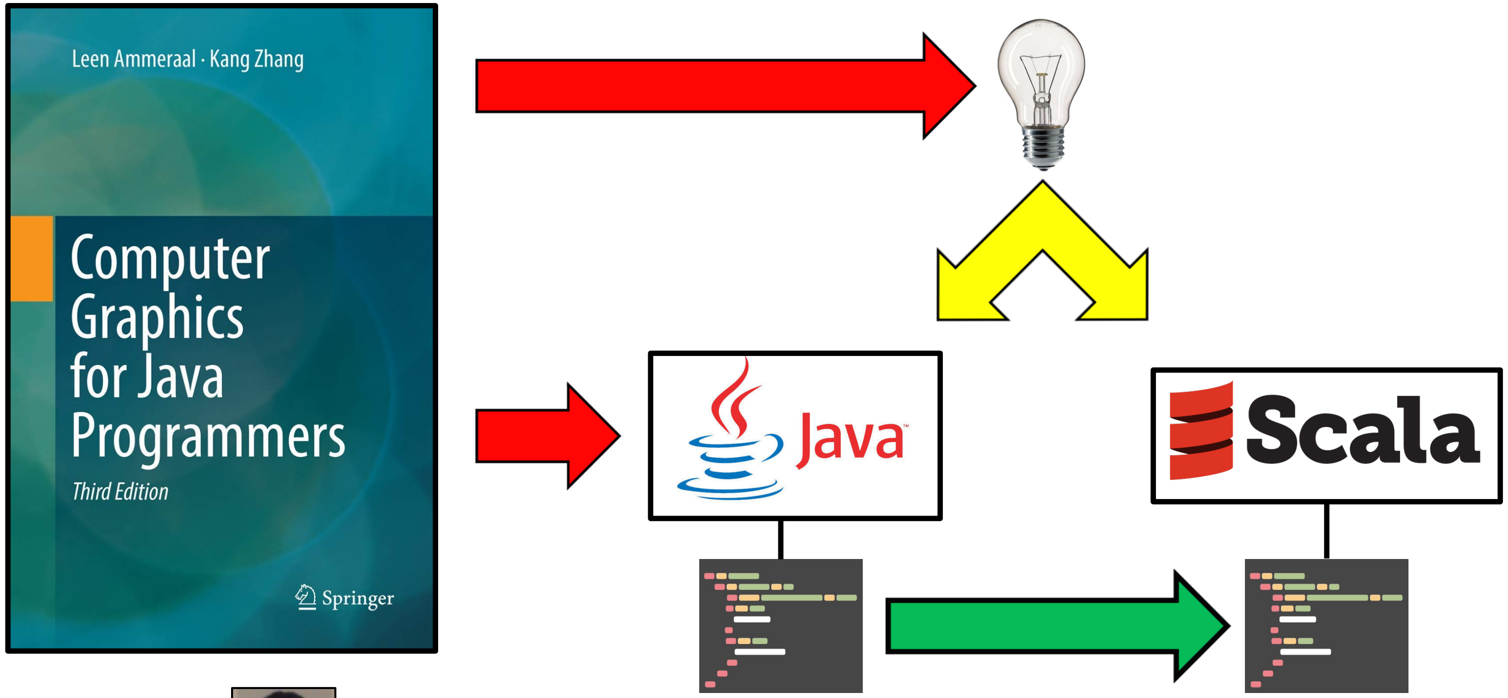
Computer Graphics in Java and Scala

Part 1

Continuous (**Logical**) and **Discrete** (**Device**) Coordinates

with a simple yet pleasing example involving concentric triangles



slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

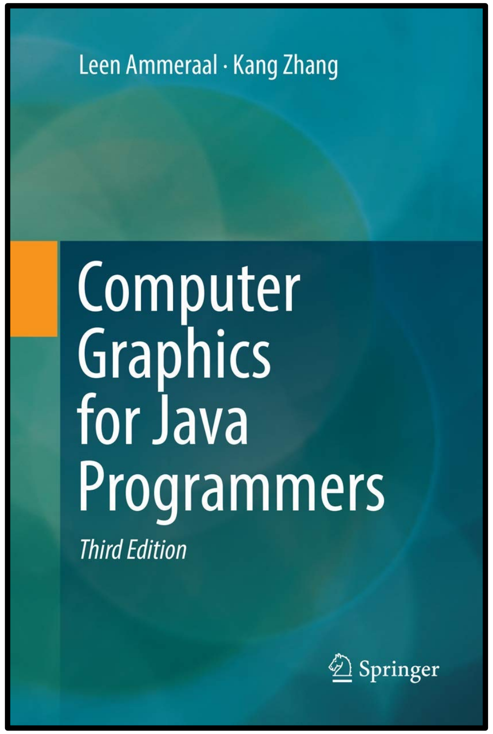
The idea of this series of decks is to have fun going through selected topics in books like **Computer Graphics for Java Programmers** in order to

- learn (or reacquaint ourselves with) some well established **computer graphics techniques**
- see some of the **Java** code that the book uses to illustrate the techniques
- rewrite the code in **Scala**, hopefully encountering opportunities to use some **functional programming** techniques

The subject of this first deck is

- the distinction between **continuous** (**logical**) and **discrete** (**device**) **coordinates**
- an example of using the technique to **draw** an **interesting pattern** involving **triangles**

@philip_schwarz



Leen Ammeraal

https://www.linkedin.com/in/leen-ammeraal-b97b968/

Kang Zhang

https://profiles.utdallas.edu/kang.zhang

The following program lines in the paint method show how to obtain the **canvas dimensions** and how to interpret them:

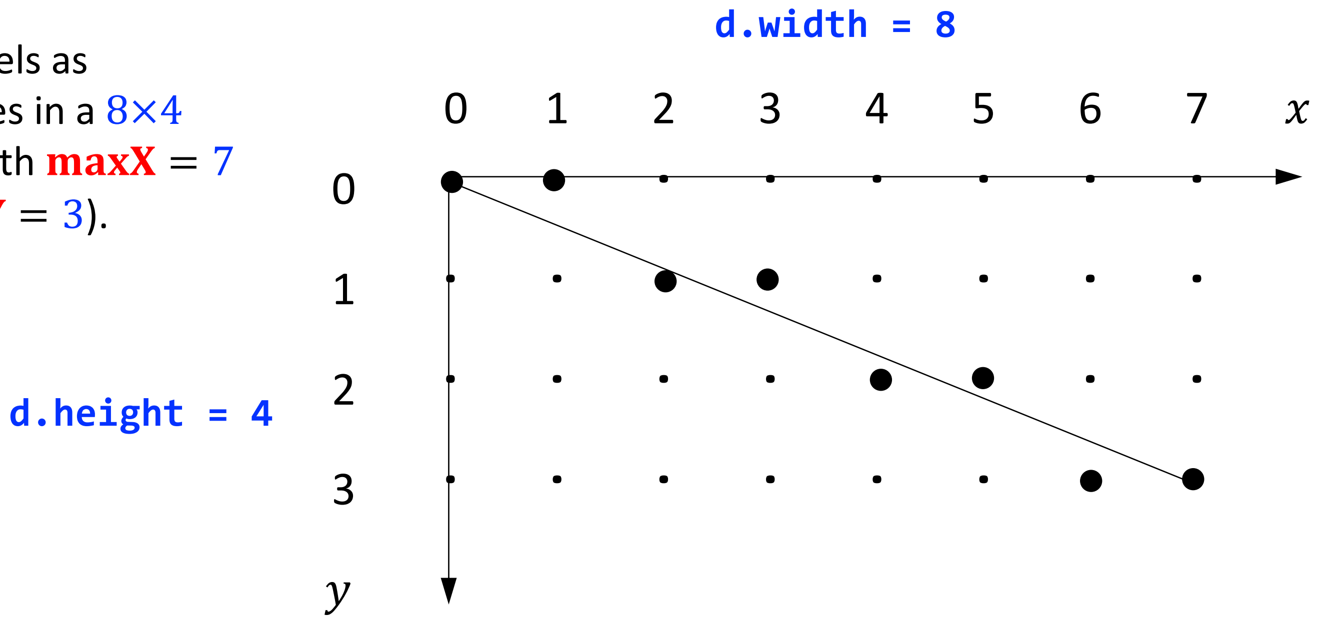
```
Dimension d = getSize();
int maxX = d.width - 1;
int maxY = d.height - 1;
```

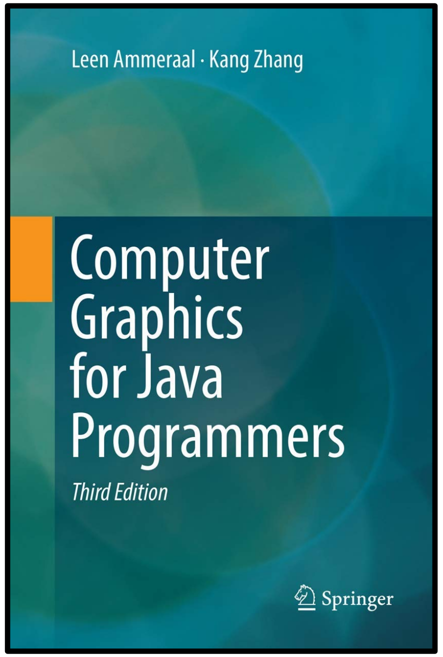
The **getSize** method of ***Component*** (a superclass of ***Canvas***) supplies us with the numbers of **pixels** on **horizontal** and **vertical** lines of the **canvas**. Since we start counting at zero, the highest **pixel** numbers, **maxX** and **maxY**, on these lines are one less than these numbers of **pixels**.

…

Figure 1.2 illustrates this for a very small **canvas**, which is only 8 **pixels** wide and 4 **pixels** high, showing a much enlarged screen grid structure. It also shows that the line connecting the **points** $(0,0)$ and $(7,3)$ is approximated by a set of eight **pixels**.

Fig 1.2 Pixels as coordinates in a 8×4 canvas (with $\mathbf{maxX} = 7$ and $\mathbf{maxY} = 3$).





1.2 Logical Coordinates

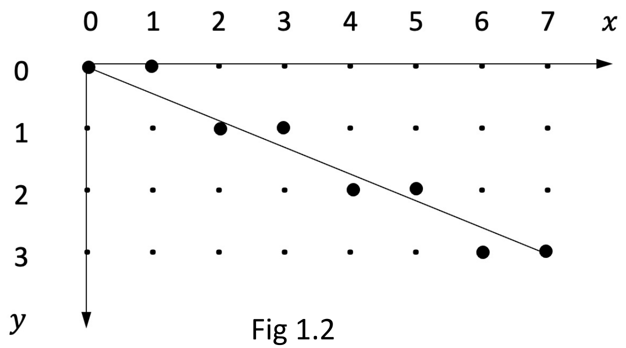


Fig 1.2

The Direction of the y-axis

As Fig 1.2 shows, the origin of the **device-coordinate systems** lies at the top-left corner of the **canvas**, so that the positive **y-axis** points downward.

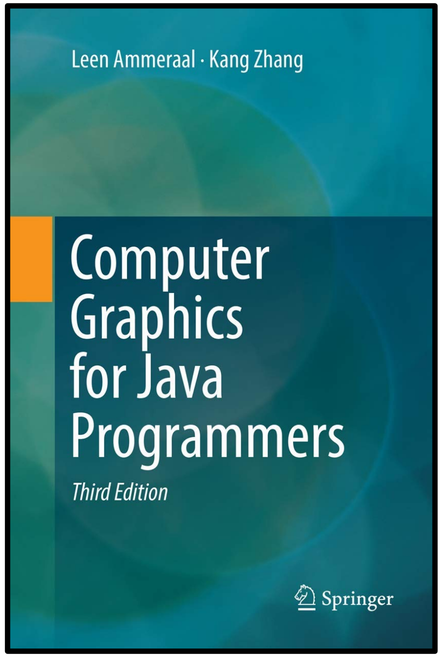
This is reasonable for text output, that starts at the top and increases y as we go to the next line of text.

However, this **direction** of the **y-axis** is different from typical mathematical practice and therefore **often inconvenient** in graphics applications.

For example, in a discussion about a line with a positive slope, we expect to go upward when moving along this line from left to right.

Fortunately we can arrange for the **positive** y **direction** to be **reversed** by performing this simple **transformation**:

$$y' = maxY - y$$



Continuous Versus Discrete Coordinates

Instead of the **discrete** (**integer**) **coordinates** at the **lower**, **device-oriented level**, we often wish to use **continuous** (**floating-point**) **coordinates** at the **higher**, **problem-oriented level**. Other useful terms are ***device*** and ***logical*** coordinates, respectively.

Writing **conversion routines** to compute **device coordinates** from the corresponding **logical** ones and vice versa is a little bit tricky. We must be aware that there are **two solutions** to this problem: **rounding** and **truncating**, even in the simple case in which increasing a **logical coordinate** by one results in increasing the **device coordinate** also by one. We wish to write the following methods:

$iX(x), iY(y)$: *converting the **logical coordinates** **x** and **y** to **device coordinates**;*
$fx(X), fy(Y)$: *converting the **device coordinates** **X** and **Y** to **logical coordinates**.*

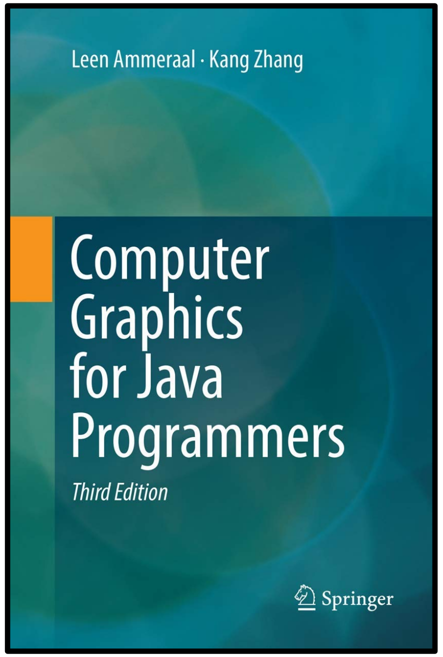
One may notice that we have used **lower-case letters** to represent **logical coordinates** and **capital letters** to represent **device coordinates**. This will be the convention used throughout this book. With regard to x-coordinates, the **rounding solution** could be:

```
int iX(float x) { return Math.round(x); }
float fx(int x) { return (float)x; }
```

For example, with this solution we have:

$$iX(2.8) = 3$$
$$fx(3) = 3.0$$

The **i** in iX is due to the function returning an **int**. Similarly for fx, which returns a **float**.



The **truncating solution** could be:

```
int iX(float x) { return (int)x; }           // Not used in
float fx(int x) { return (float)x + 0.5F; } // this book
```

With these **conversion functions**, we would have

$$iX(2.8) = 2$$
$$fx(2) = 2.5$$

We will use the **rounding solution** throughout this book, since it is the **better choice** if **logical coordinates** frequently happen to be integer values. In these cases the practice of **truncating** floating-point numbers will often lead to **worse results** than those with **rounding**.

Apart from the above methods iX and fx (based on rounding), for **x-coordinates**, we need similar methods for **y-coordinates**, taking into account the opposite direction of the two **y-axes**. At the bottom of the **canvas**, the **device y-coordinate** is **maxY**, while the **logical y-coordinate** is 0, which may explain the two expressions of the form **maxY - …** in the following methods:

```
int iX(float x) { return Math.round(x); }
int iY(float y) { return maxY - Math.round(y); }
float fx(int x) { return (float)x; }
float fy(int y) { return (float)(maxY - y); }
```

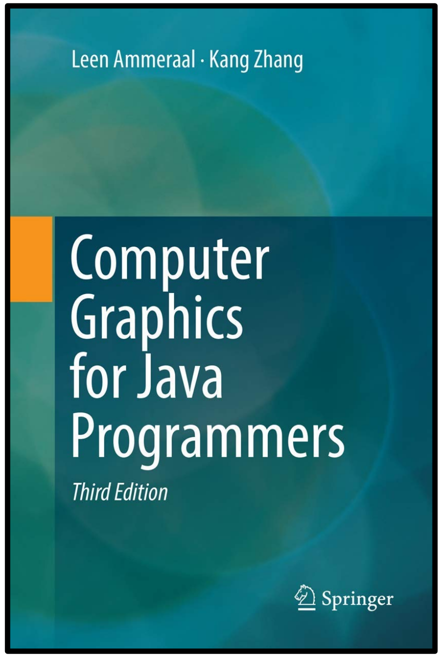


Figure 1.4 shows a fragment of a **canvas**, based on **maxY**=16.
The **pixels** are drawn as black dots, each placed at the center of a square of dashed lines, and the **device coordinates** (X,Y) are placed between parentheses near each dot. For example, the **pixel** with **device coordinates (8,2)** at the upper-right corner of this **canvas** fragment, has **logical coordinates (8.0, 14.0)**. We have

```
iX(8.0) = Math.round(8.0) = 8
iY(14.0) = 16 - Math.round(14.0) = 2
fx(8) = (float)8 = 8.0
fy(2) = (float)(16 - 2) = 14.0
```

logical

device

The dashed square around this dot denotes all points (x,y) satisfying:

$$7.5 \le x < 8.5$$
$$13.5 \le y < 14.5$$

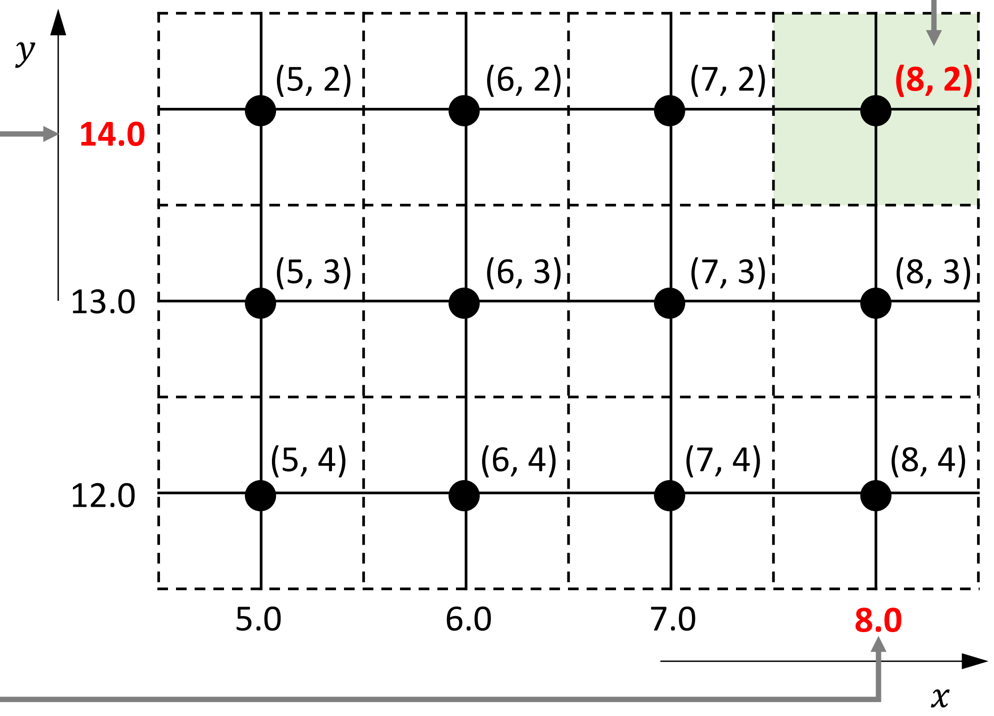
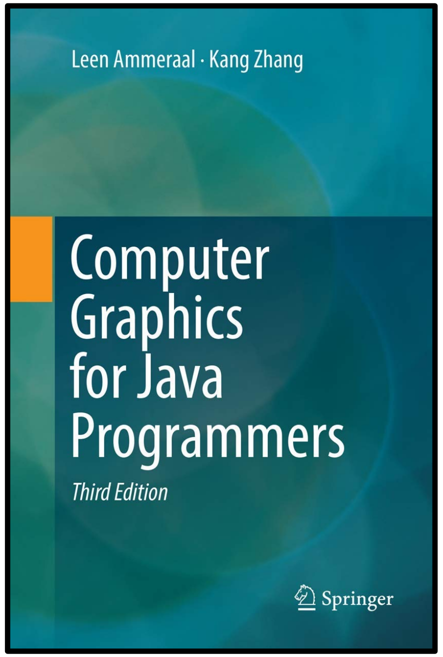


Figure 1.4 Logical and device coordinates, based on maxY = 16

All these points are converted to the **pixel** (8,2) by our methods iX and iY. Let us demonstrate this way of converting **floating-point logical coordinates** to **integer device coordinates** in a program that begins by drawing an **equilateral triangle ABC**, with the side **AB** at the bottom and the point **C** at the top. Then using

$$q = 0.05$$
$$p = 1 - q = 0.95$$

We compute the new points **A'**, **B'** and **C'** near **A**, **B** and **C**



and lying on the sides **AB**, **BC** and **CA** respectively, writing:

```
xA1 = p * xA + q * xB;
yA1 = p * yA + q * yB;
xB1 = p * xB + q * xC;
yB1 = p * yB + q * yC;
xC1 = p * xC + q * xA;
yC1 = p * yC + q * yA;
```

We then draw the triangle **A'B'C'**, which is slightly smaller than **ABC** and turned a little **counter-clockwise**. Applying the same principle to triangle **A'B'C'** to obtain a third triangle **A''B''C''**, and so on, until 50 triangles have been drawn, the result will be as shown in Fig 1.5. If we change the dimensions of the **window**, new **equilateral triangles** appear, again in the center of the **canvas** and with dimensions proportional to the size of this **canvas**.

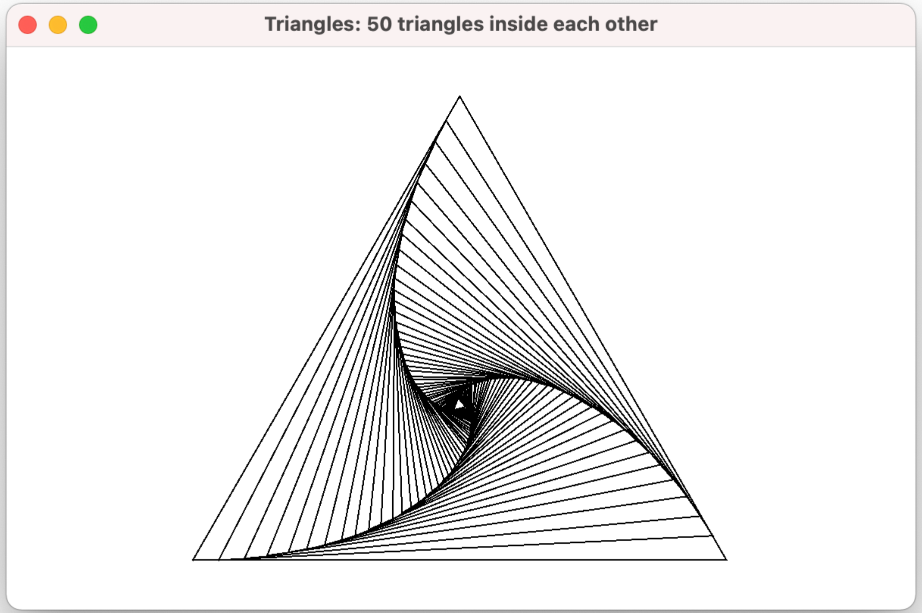
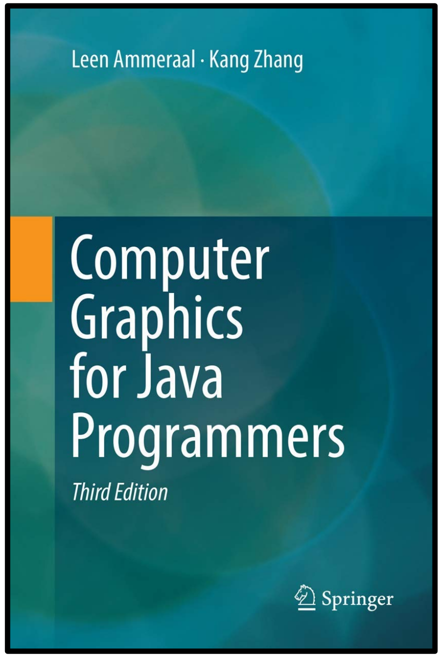


Figure 1.5 Triangles, drawn inside each other



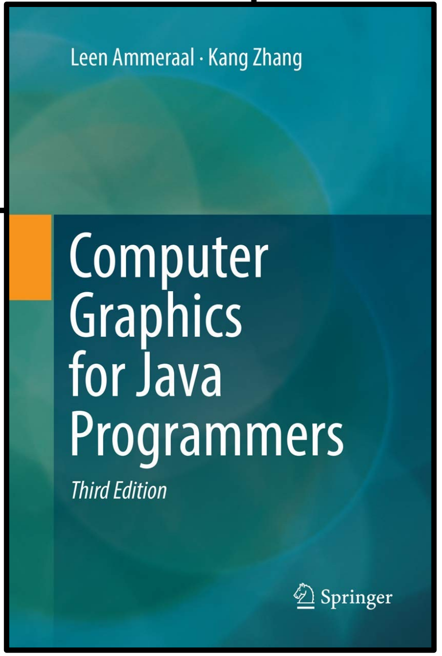
@philip_schwarz

The next slide shows the **Java** code for the whole program.

```java
import java.awt.*;

public class CvTriangles extends Canvas {
  int maxX, maxY, minMaxXY, xCenter, yCenter;

  void initgr() {
    Dimension d = getSize();
    maxX = d.width - 1; maxY = d.height - 1;
    minMaxXY = Math.min(maxX, maxY);
    xCenter = maxX / 2; yCenter = maxY / 2;
  }

  int iX(float x) { return Math.round(x); }
  int iY(float y) { return maxY - Math.round(y); }

  public void paint(Graphics g) {
    initgr();
    float side = 0.95F * minMaxXY, sideHalf = 0.5F * side,
          h = sideHalf * (float) Math.sqrt(3),
          xA, yA, xB, yB, xC, yC, xA1, yA1, xB1, yB1, xC1, yC1, p, q;
    q = 0.05F; p = 1 - q;
    xA = xCenter - sideHalf; yA = yCenter - 0.5F * h;
    xB = xCenter + sideHalf; yB = yA;
    xC = xCenter; yC = yCenter + 0.5F * h;
    for (int i = 0; i < 50; i++) {
      g.drawLine(iX(xA), iY(yA), iX(xB), iY(yB));
      g.drawLine(iX(xB), iY(yB), iX(xC), iY(yC));
      g.drawLine(iX(xC), iY(yC), iX(xA), iY(yA));
      xA1 = p * xA + q * xB; yA1 = p * yA + q * yB;
      xB1 = p * xB + q * xC;  yB1 = p * yB + q * yC;
      xC1 = p * xC + q * xA;  yC1 = p * yC + q * yA;
      xA = xA1; xB = xB1; xC = xC1;
      yA = yA1; yB = yB1; yC = yC1;
    }
  }
}
```

```java
// Triangles.java: This program draws 50
// triangles inside each other.
public class Triangles extends Frame {

  public static void main(String[] args) {
    new Triangles();
  }

  Triangles() {
    super("Triangles: 50 triangles inside each other");
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    setSize(600, 400);
    add("Center", new CvTriangles());
    setVisible(true);
  }
}
```

Without **floating-point** **logical coordinates** and with a **y-axis** pointing downward, this program would have been less easy to write.



Let's rewrite that Java code in Scala, beginning with the code section which given a starting triangle, draws the triangle and then proceeds to repeatedly, first shrink and twist the triangle, and then draw it, thus generating and drawing 49 more concentric triangles.

While the Java code uses a java.awt.Canvas, the Scala code uses a javax.swing.JPanel.

```java
public class CvTriangles extends Canvas {
  …
  public void paint(Graphics g) {
    …
    for (int i = 0; i < 50; i++) {
      g.drawLine(iX(xA), iY(yA), iX(xB), iY(yB));
      g.drawLine(iX(xB), iY(yB), iX(xC), iY(yC));
      g.drawLine(iX(xC), iY(yC), iX(xA), iY(yA));
      xA1 = p * xA + q * xB; yA1 = p * yA + q * yB;
      xB1 = p * xB + q * xC;  yB1 = p * yB + q * yC;
      xC1 = p * xC + q * xA;  yC1 = p * yC + q * yA;
      xA = xA1; xB = xB1; xC = xC1;
      yA = yA1; yB = yB1; yC = yC1;
    }
  }
}
```

```scala
class TrianglesPanel extends JPanel:
  …
  override def paintComponent(g: Graphics): Unit =
    …
    LazyList.iterate(triangle)(shrinkAndTwist).take(50).foreach(draw)
```

```scala
shrinkAndTwist: Triangle => Triangle
```

```scala
draw: Triangle => Unit
```

Given an initial triangle, we are going to generate a lazy, potentially infinite, sequence of triangles, in which each triangle, with the exception of the first one, is the result of shrinking and twisting the previous triangle.

We then take (materialise) the first 50 triangles of the sequence and iterate through them, drawing each one in turn.

A triangle consists of three **logical points** (points with **logical coordinates**) A, B and C.

```scala
case class Point(x: Float, y: Float)
case class Triangle(a: Point, b: Point, c: Point)
```

Drawing a triangle amounts to **drawing lines** AB, BC and CA.

```scala
val draw: Triangle => Unit =
  case Triangle(a, b, c) =>
    drawLine(a, b)
    drawLine(b, c)
    drawLine(c, a)
```

```scala
LazyList.iterate(triangle)(shrinkAndTwist).take(50).foreach(draw)
```

To draw a line from **logical point** A to **logical point** B, we first compute the **coordinates** of the corresponding **device points**, and then pass those coordinates to the **drawLine** method provided by the **Graphics** object.

```scala
def drawLine(a: Point, b: Point): Unit =
  val (ax,ay) = a.deviceCoords(panelHeight)
  val (bx,by) = b.deviceCoords(panelHeight)
  g.drawLine(ax, ay, bx, by)
```

Here is how we enrich a **logical point** with a **deviceCoords** function that takes the **logical coordinates** of the point and computes the corresponding **device coordinates**:

```scala
extension (p: Point)
  def deviceCoords(panelHeight: Int): (Int, Int) =
    (Math.round(p.x), panelHeight - Math.round(p.y))
```

The **deviceCoords** function is our **Scala** equivalent of **Java** functions iX and iY:

```java
int iX(float x) { return Math.round(x); }
int iY(float y) { return maxY - Math.round(y); }
```

As for the Java code that **shrinks** and **twists** a **triangle**, in our Scala code, we encapsulate it in function **shrinkAndTwist**.

@philip_schwarz

```java
float …
      xA, yA, xB, yB, xC, yC, xA1,
      yA1, xB1, yB1, xC1, yC1,
      p, q;
```

```java
q = 0.05F; p = 1 - q;
```

```java
  xA1 = p * xA + q * xB; yA1 = p * yA + q * yB;
  xB1 = p * xB + q * xC;  yB1 = p * yB + q * yC;
  xC1 = p * xC + q * xA;  yC1 = p * yC + q * yA;
  xA = xA1; xB = xB1; xC = xC1;
  yA = yA1; yB = yB1; yC = yC1;
```

```scala
val shrinkAndTwist: Triangle => Triangle =
  val q = 0.05F
  val p = 1 - q
  def combine(a: Point, b: Point) = Point(p * a.x + q * b.x, p * a.y + q * b.y)
  { case Triangle(a,b,c) => Triangle(combine(a,b), combine(b,c), combine(c,a)) }
```

As for the Java code that computes the first triangle from the dimensions of the Canvas on which it is going to be drawn, here it is, together with the corresponding Scala code, which draws on a JPanel.

```java
int maxX, maxY, minMaxXY, xCenter, yCenter;

void initgr() {
  Dimension d = getSize();
  maxX = d.width - 1; maxY = d.height - 1;
  minMaxXY = Math.min(maxX, maxY);
  xCenter = maxX / 2; yCenter = maxY / 2;
}
```

```java
float side = 0.95F * minMaxXY, sideHalf = 0.5F * side,
      h = sideHalf * (float) Math.sqrt(3),
      …;
```

```java
xA = xCenter - sideHalf; yA = yCenter - 0.5F * h;
xB = xCenter + sideHalf; yB = yA;
xC = xCenter; yC = yCenter + 0.5F * h;
```

```scala
val panelSize: Dimension = getSize()
val panelWidth = panelSize.width - 1
val panelHeight = panelSize.height - 1
val panelCentre = Point(panelWidth / 2, panelHeight / 2)
val triangleSide = 0.95F * Math.min(panelWidth, panelHeight)
val triangleHeight = (0.5F * triangleSide) * Math.sqrt(3).toFloat
```

```scala
object Triangle:
  def apply(centre: Point, side: Float, height: Float): Triangle =
    val Point(x,y) = centre
    val halfSide = 0.5F * side
    val bottomLeft = Point(x - halfSide, y - 0.5F * height)
    val bottomRight = Point(x + halfSide, y - 0.5F * height)
    val top = Point(x, y + 0.5F * height )
    Triangle(bottomLeft,bottomRight,top)
```

```scala
val triangle = makeTriangle(panelCentre,
                            triangleSide,
                            triangleHeight)
```

And finally, let's translate the rest of the code, which creates the application's frame.

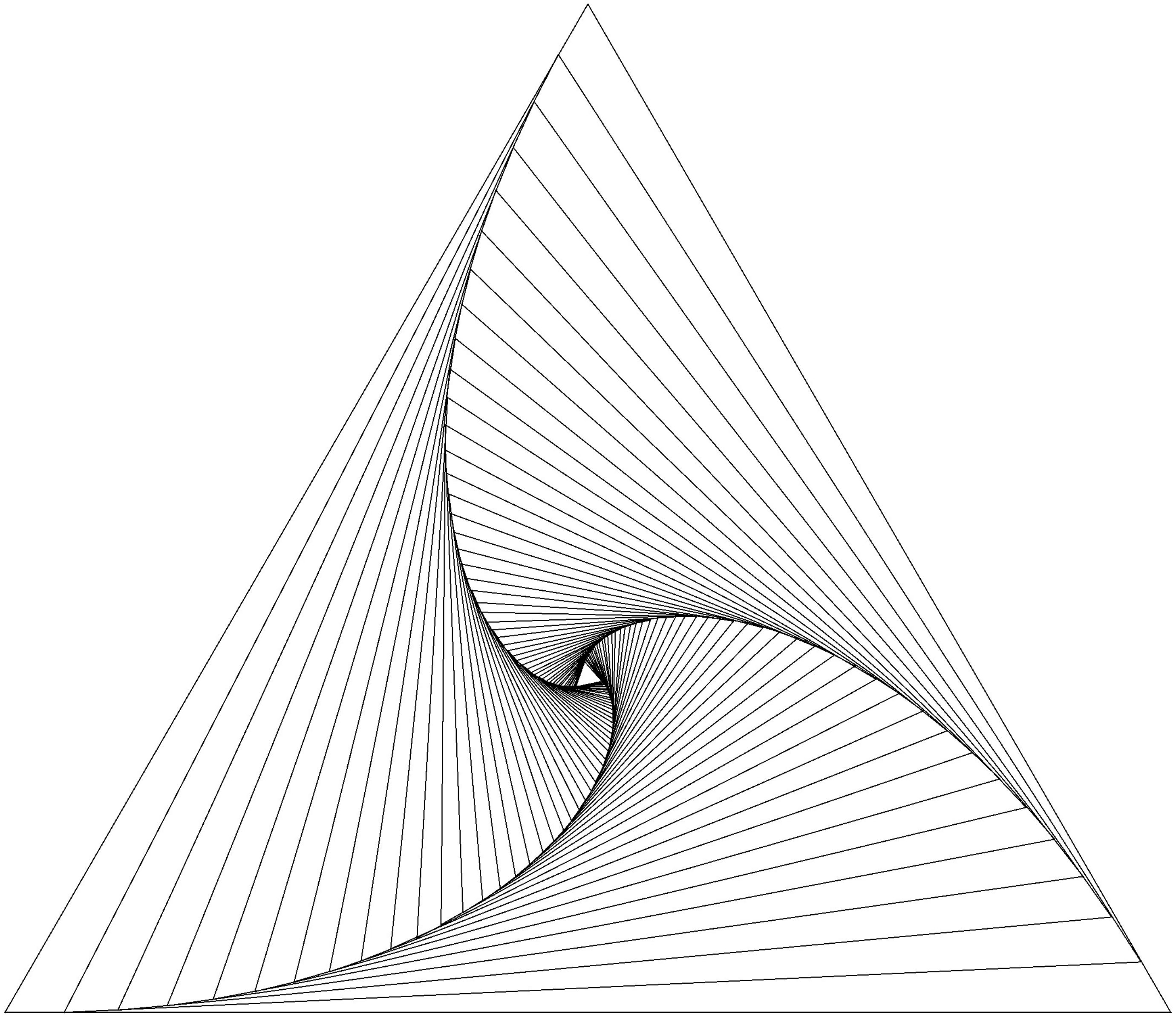
```java
public class Triangles extends Frame {

  public static void main(String[] args) {
    new Triangles();
  }

  Triangles() {
    super("Triangles: 50 triangles inside each other");
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    setSize(600, 400);
    add("Center", new CvTriangles());
    setVisible(true);
  }
}
```

```scala
class Triangles:
  JFrame.setDefaultLookAndFeelDecorated(true)
  val frame = new JFrame("Triangles: 50 triangles inside each other")
  frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE)
  frame.setSize(600, 400)
  frame.add(TrianglesPanel())
  frame.setVisible(true)
```

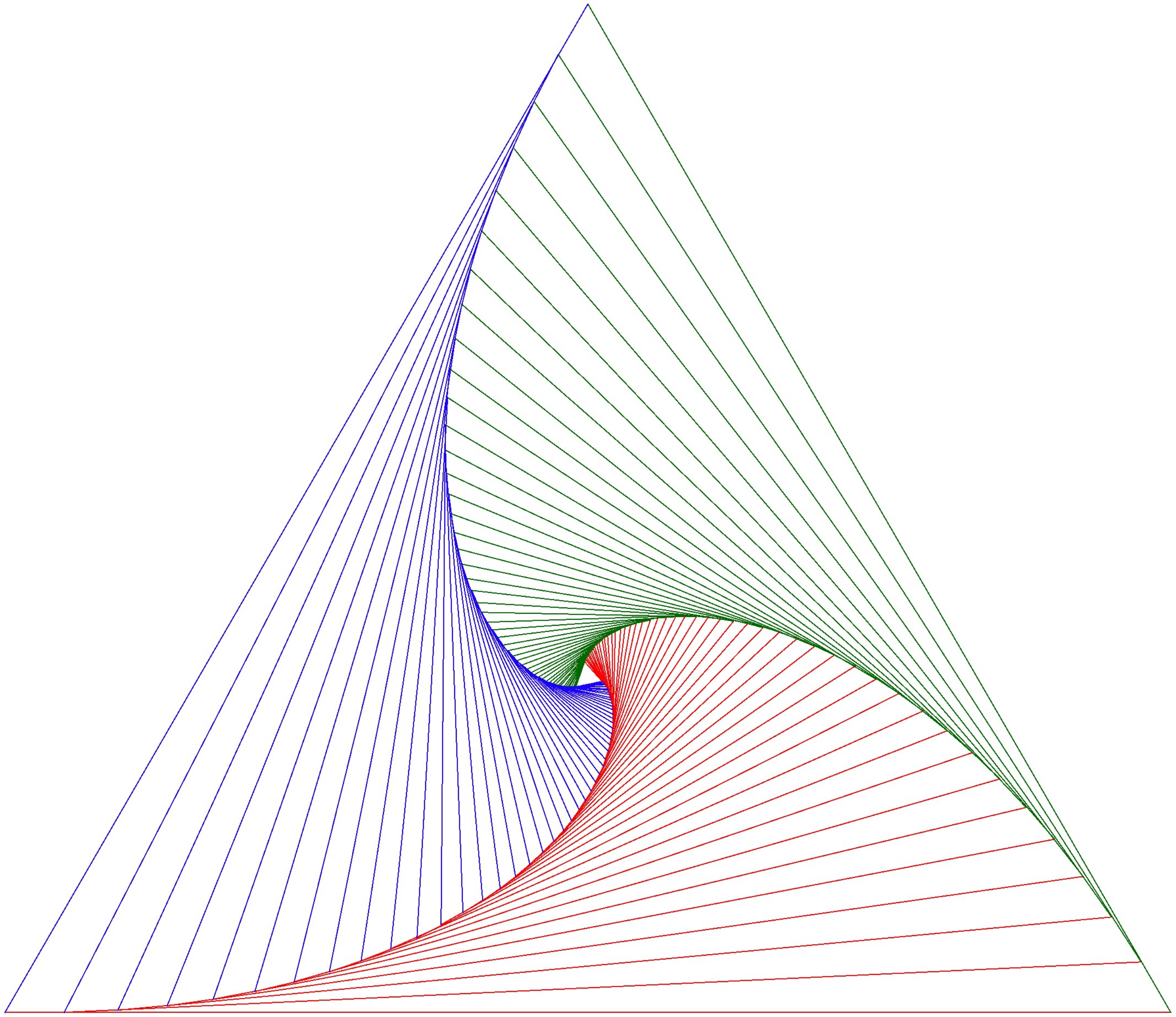
```scala
@main def main: Unit =
  // Create a panel on the event dispatching thread
  SwingUtilities.invokeLater(
    new Runnable():
      def run: Unit = Triangles()
  )
```

Now let's run the **Scala** program, to verify that it works OK.

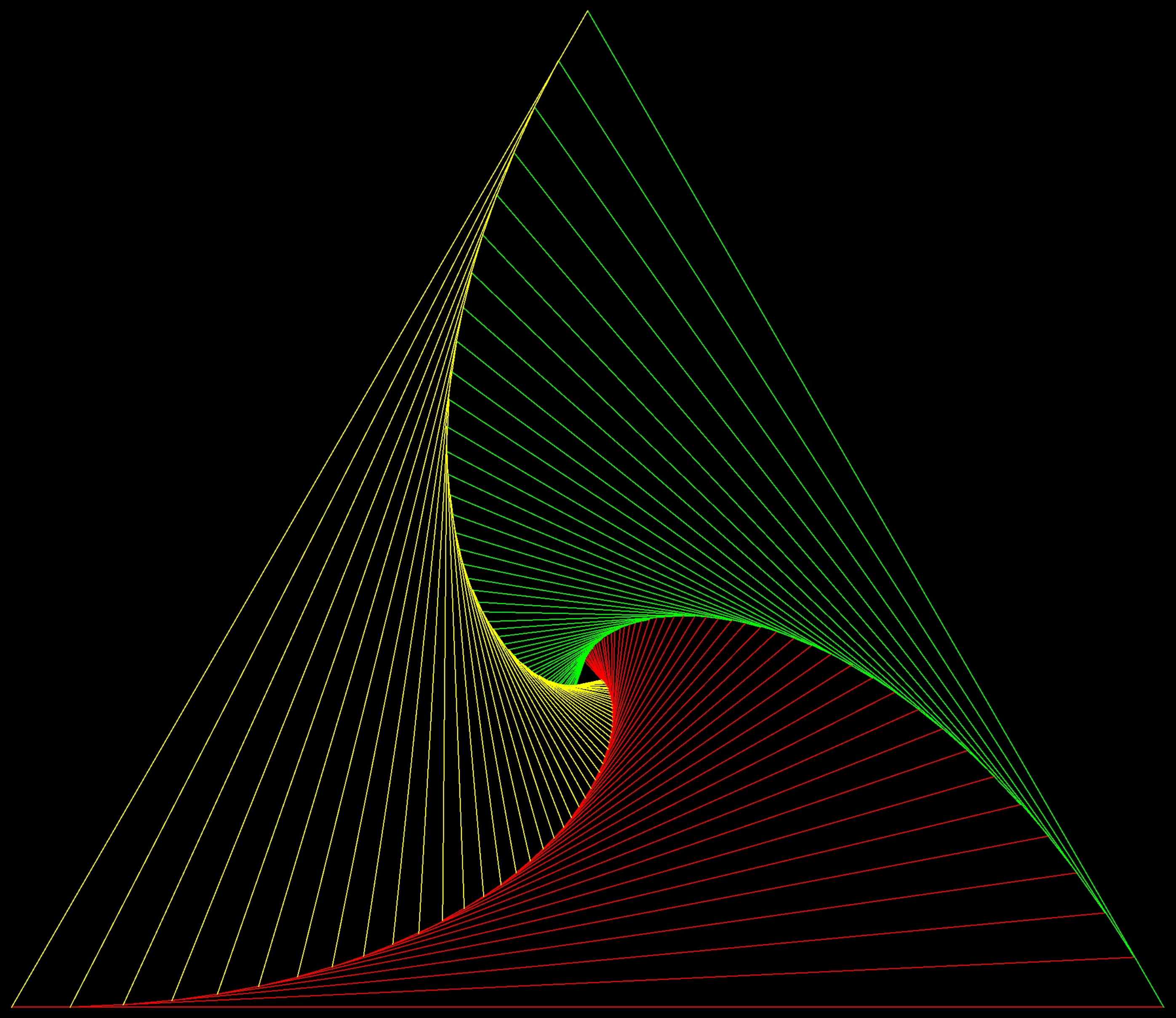
@philip_schwarz



Now let's run the program again after temporarily tweaking it so that each side of a triangle is drawn in a different colour.



And now let’s repeat that,
but with different colours
and a black background.



The next slide shows our **Scala** code in its entirety, and the subsequent slide shows the whole of the **Java** code again, for comparison.

```scala
class TrianglesPanel extends JPanel:

  setBackground(Color.white)

  override def paintComponent(g: Graphics): Unit =

    super.paintComponent(g)

    val panelSize: Dimension = getSize()
    val panelWidth = panelSize.width - 1
    val panelHeight = panelSize.height - 1
    val panelCentre = Point(panelWidth / 2, panelHeight / 2)
    val triangleSide: Float = 0.95F * Math.min(panelWidth, panelHeight)
    val triangleHeight: Float = (0.5F * triangleSide) * Math.sqrt(3).toFloat

    val shrinkAndTwist: Triangle => Triangle =
      val q = 0.05F
      val p = 1 - q
      def combine(a: Point, b: Point) = Point(p * a.x + q * b.x, p * a.y + q * b.y)
      { case Triangle(a,b,c) => Triangle(combine(a,b), combine(b,c), combine(c,a)) }

    def drawLine(a: Point, b: Point): Unit =
      val (ax,ay) = a.deviceCoords(panelHeight)
      val (bx,by) = b.deviceCoords(panelHeight)
      g.drawLine(ax, ay, bx, by)

    val draw: Triangle => Unit =
      case Triangle(a, b, c) =>
        drawLine(a, b)
        drawLine(b, c)
        drawLine(c, a)

    val triangle = Triangle(panelCentre, triangleSide, triangleHeight)

    LazyList.iterate(triangle)(shrinkAndTwist).take(50).foreach(draw)
```

```scala
case class Point(x: Float, y: Float)
```

```scala
extension (p: Point)
  def deviceCoords(panelHeight: Int): (Int, Int) =
    (Math.round(p.x), panelHeight - Math.round(p.y))
```

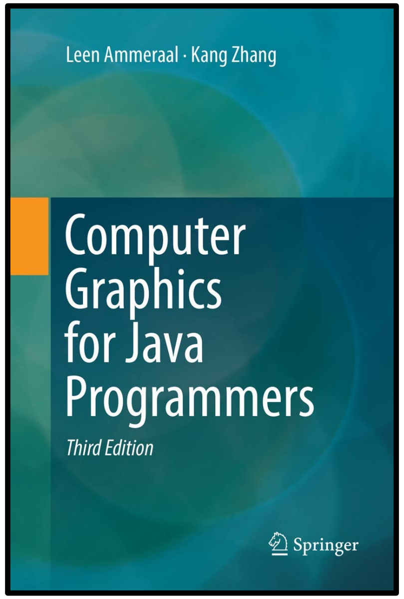
```scala
case class Triangle(a: Point, b: Point, c: Point)
```

```scala
object Triangle:
  def apply(centre: Point, side: Float, height: Float): Triangle =
    val Point(x,y) = centre
    val halfSide = 0.5F * side
    val bottomLeft = Point(x - halfSide, y - 0.5F * height)
    val bottomRight = Point(x + halfSide, y - 0.5F * height)
    val top = Point(x, y + 0.5F * height )
    Triangle(bottomLeft,bottomRight,top)
```

```scala
class Triangles:
  JFrame.setDefaultLookAndFeelDecorated(true)
  val frame =
    new JFrame("Triangles: 50 triangles inside each other")
  frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE)
  frame.setSize(600, 400)
  frame.add(TrianglesPanel())
  frame.setVisible(true)
```

```scala
@main def main: Unit =
  // Create a panel on the event dispatching thread
  SwingUtilities.invokeLater(
    new Runnable():
      def run: Unit = Triangles()
  )
```

```java
// Triangles.java: This program draws 50
// triangles inside each other.
public class Triangles extends Frame {

  public static void main(String[] args) {
    new Triangles();
  }

  Triangles() {
    super("Triangles: 50 triangles inside each other");
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    setSize(600, 400);
    add("Center", new CvTriangles());
    setVisible(true);
  }
}
```




```java
public class CvTriangles extends Canvas {
  int maxX, maxY, minMaxXY, xCenter, yCenter;

  void initgr() {
    Dimension d = getSize();
    maxX = d.width - 1; maxY = d.height - 1;
    minMaxXY = Math.min(maxX, maxY);
    xCenter = maxX / 2; yCenter = maxY / 2;
  }

  int iX(float x) { return Math.round(x); }
  int iY(float y) { return maxY - Math.round(y); }

  public void paint(Graphics g) {
    initgr();
    float side = 0.95F * minMaxXY, sideHalf = 0.5F * side,
          h = sideHalf * (float) Math.sqrt(3),
          xA, yA, xB, yB, xC, yC, xA1, yA1, xB1, yB1, xC1, yC1, p, q;
    q = 0.05F; p = 1 - q;
    xA = xCenter - sideHalf; yA = yCenter - 0.5F * h;
    xB = xCenter + sideHalf; yB = yA;
    xC = xCenter; yC = yCenter + 0.5F * h;
    for (int i = 0; i < 50; i++) {
      g.drawLine(iX(xA), iY(yA), iX(xB), iY(yB));
      g.drawLine(iX(xB), iY(yB), iX(xC), iY(yC));
      g.drawLine(iX(xC), iY(yC), iX(xA), iY(yA));
      xA1 = p * xA + q * xB; yA1 = p * yA + q * yB;
      xB1 = p * xB + q * xC;  yB1 = p * yB + q * yC;
      xC1 = p * xC + q * xA;  yC1 = p * yC + q * yA;
      xA = xA1; xB = xB1; xC = xC1;
      yA = yA1; yB = yB1; yC = yC1;
    }
  }
}
```

That's all for now.
See you in Part 2.

@philip_schwarz